title: Sending Bundles

Getting our trade ready

After we find a transaction that touches the trading pair we're targeting, we need to calculate how many tokens we should expect to receive. In this project, we are specifically *buying* DAI with ETH (technically WETH, because Uniswap only trades ERC-20 tokens), so we want the price of DAI/WETH to be as low as possible. In code, we define our price requirements in terms of the *amount of tokens we receive* from the trade — so we want the *highest* amount of tokens possible, but we have a definite minimum (1800 DAI).

In terms of a limit order, we're saying that we want our order to be filled when the price is *at most* 1800 DAI, but if the price is lower, then we want to fill at the lower price, which yields more DAI per WETH.

To find our token's market price, we simulate our trade by calling the swapExactTokensForTokens function with a static call. A static call simply simulates the transaction, so we can see what it would do if we were to actually send it. We set our buy/sell amounts that we defined earlier to see how much we'd get from the swap. Add this function to your code — we'll add it to our main function later.

src/index.ts

tsx async function getBuyTokenAmountWithExtra() { const resultCallResult = await uniswapRouterContract .swapExactTokensForTokens .staticCallResult(SELL_TOKEN_AMOUNT, 1n, [SELL_TOKEN_ADDRESS, BUY_TOKEN_ADDRESS], executorWallet.address, 9999999999n) const normalOutputAmount = resultCallResult[0][1] const extraOutputAmount = normalOutputAmount * (10000n + DISCOUNT_IN_BPS) / 10000n return extraOutputAmount }

The minimum amount we can expect to receive from a swap is defined by normalOutputAmount. Then, we calculate how much we'd get if with a 40 basis-points discount, which we should expect if we successfully backrun a transaction that shifts the price in our favor, and assign this value to extraOutputAmount.

When we detect a new transaction, we'll need to check the going price and set up our trade accordingly. If the price is lower than our target, and because we're trying to *buy* tokens, we want to make sure our trade expects more tokens out; as many as we can get at the lower price with our fixed sell amount (the ETH we'll spend to buy the tokens). If the price is higher than our target, then we'll just set the expected output to the minimum amount we'd expect to if the price were at our target, in hopes that the transaction we backrun will move the price enough for us to make a trade.

Let's add a couple more functions to implement this logic:

src/index.ts

```tsx async function getSignedBackrunTx( outputAmount: bigint, nonce: number ) { const backrunTx = await uniswapRouterContract.swapExactTokensForTokens.populateTransaction(SELL_TOKEN_AMOUNT, outputAmount, [SELL_TOKEN_ADDRESS, BUY_TOKEN_ADDRESS], executorWallet.address, 9999999999n) const backrunTxFull = { ...backrunTx, chainId: 1, maxFeePerGas: MAX_GAS_PRICE * GWEI, maxPriorityFeePerGas: MAX_PRIORITY_FEE * GWEI, gasLimit: TX_GAS_LIMIT, nonce: nonce } return executorWallet.signTransaction(backrunTxFull) }

async function backrunAttempt( currentBlockNumber: number, nonce: number, pendingTxHash: string ) { let outputAmount = await getBuyTokenAmountWithExtra() if (outputAmount < BUY_TOKEN_AMOUNT_CUTOFF) { console.log(Even with extra amount, not enough BUY token: ${ outputAmount.toString() }. Setting to amount cut-off) outputAmount = BUY_TOKEN_AMOUNT_CUTOFF } const backrunSignedTx = await getSignedBackrunTx(outputAmount, nonce) try { const sendBundleResult = await mevshare.sendBundle({ inclusion: { block: currentBlockNumber + 1 }, body: [ { hash: pendingTxHash }, { tx: backrunSignedTx, canRevert: false } ] },) console.log('Bundle Hash: ' + sendBundleResult.bundleHash) } catch (e) { console.log('err', e) } } ```

The getSignedBackrunTx function creates the transaction we'll send to execute our trade on Uniswap. We set a fixed gas price here for simplicity. If you prefer, you could replace the with dynamic fees that track the base fee of the chain. But constant gas prices may work better if you don't want to spend a lot on gas, and don't mind having to wait if the network's base fee exceeds your settings.

The backrunAttempt function defines our price requirement logic: we make sure that outputAmount is at least our previously-defined cutoff amount. However, if the simulated output amount is higher, then we set outputAmount to expect that much, which protects us from slippage in case other transactions in the block happen to trade on the same pair. This function then sends our bundle to MEV-Share. If the bundle was received successfully, we should see a bundle hash logged to our console.

Using getBuyTokenAmountWithExtra, we define outputAmount, the amount we expect to receive from the trade. We create our backrun transaction backrunSignedTx and send it to MEV-Share in a bundle by calling mevshare.sendBundle.

*Real quick, let's break down the bundle we passed to sendBundle:*
```

The `block` parameter in `inclusion` specifies which block we want the bundle to land in. We indicate here that we want our bundle to land in the next block.

The `body` parameter is where we set our bundle's transactions. The order in which they're specified is the order in which they'll execute on chain. Each transaction is specified as an object, with either a `hash` parameter, or a `tx` parameter (paired with `canRevert` to specify whether this transaction is allowed to revert and land on chain). The transaction we specify with `hash` is the pending transaction from the event stream that we want to backrun. We have to use its hash because the MEV-Share event stream does not reveal the entire signed transaction. Naturally, the following transaction, specified by `tx`, is our trade.

Once we stitch all these new functions into our main loop, our bot will be done!

Sending a backrun bundle

When we detect a new pending transaction in the `mevshare.on("transaction")` callback that affects the price of our target pair, we need to send a bundle using the `backrunAttempt` function. This bundle checks our target price and sets up our trade to get us the best price possible.

src/index.ts

```
tsx mevshare.on("transaction", (pendingTx) => { // ... // TODO: backrun the user tx if (!transactionIsRelatedToPair(pendingTx, PAIR_ADDRESS)) { console.log('skipping tx: ' + pendingTx.hash) return } console.log(`It's a match: ${ pendingTx.hash }`) const currentBlockNumber = await provider.getBlockNumber() backrunAttempt(currentBlockNumber, nonce, pendingTx.hash) })
```

We'll also want to set up a callback that watches for new blocks and retries previous backrun attempts. Our bundles only target one block, but Protect transactions (which make up the transactions in the event stream) are valid for 25 blocks from when they're received. This means that if our backrun wasn't successful before, we can try again up to 24 more times.

Add this code to your `main` function:

```
tsx let recentPendingTxHashes: Array<{ txHash: string, blockNumber: number }> = [] provider.on('block', ( blockNumber ) => { for (const recentPendingTxHash of recentPendingTxHashes) { console.log(recentPendingTxHash) backrunAttempt(blockNumber, nonce, recentPendingTxHash.txHash) } // Cleanup old pendingTxHashes recentPendingTxHashes = recentPendingTxHashes.filter(( recentPendingTxHash ) => blockNumber > recentPendingTxHash.blockNumber + BLOCKS_TO_TRY) })
```

And in your `mevshare.on` callback, add this piece at the end:

```
tsx recentPendingTxHashes.push({ txHash: pendingTx.hash, blockNumber: currentBlockNumber })
```

When you're done, your main function should look like this:

src/index.ts

~~~
```tsx async function main() { console.log('mev-share auth address: ' + authSigner.address) console.log('executor address: ' + executorWallet.address) const PAIR_ADDRESS = (await uniswapFactoryContract.getPair(SELL_TOKEN_ADDRESS, BUY_TOKEN_ADDRESS)).toLowerCase() await approveTokenToRouter(SELL_TOKEN_ADDRESS, UNISWAP_V2_ADDRESS) const nonce = await executorWallet.getNonce('latest') let recentPendingTxHashes: Array<{ txHash: string, blockNumber: number }> = []

mevshare.on('transaction', async ( pendingTx: IPendingTransaction ) => {
  if (!transactionIsRelatedToPair(pendingTx, PAIR_ADDRESS)) {
    console.log('skipping tx: ' + pendingTx.hash)
    return
  }
  console.log(`It's a match: ${ pendingTx.hash }`)
  const currentBlockNumber = await provider.getBlockNumber()
  backrunAttempt(currentBlockNumber, nonce, pendingTx.hash)
  recentPendingTxHashes.push({ txHash: pendingTx.hash, blockNumber: currentBlockNumber })
})
provider.on('block', ( blockNumber ) => {
  for (const recentPendingTxHash of recentPendingTxHashes) {
    console.log(recentPendingTxHash)
    backrunAttempt(blockNumber, nonce, recentPendingTxHash.txHash)
  }
  // Cleanup old pendingTxHashes
  recentPendingTxHashes = recentPendingTxHashes.filter(( recentPendingTxHash ) =>
    blockNumber > recentPendingTxHash.blockNumber + BLOCKS_TO_TRY)
})

} ```
~~~

> For a full, working code example, check out https://github.com/flashbots/simple-limit-order-bot

That's all you need! This code will listen for new transactions and blocks, and trigger our code to send bundles when we find a transaction that changes the price.

Run the code and you should see something like this:

And after some time…

It may take a while to find a match — remember, the code scans for a trade on the ETH/DAI pair on Uniswap V2. There are lots of other events to consider for future improvements to this bot!

You may also consider removing the code that checks the event to see if it matches our pair address. It's possible that the price could move to our target level without us seeing an event to backrun. If we simply backrun every transaction we see, then we can potentially benefit from opportunities that we can't yet see. However, it is essential to understand how to use logs on MEV-Share, as they provide critical data that can be used for a multitude of other purposes, so we've introduced it here as a practical example.